# Application of Three Physical Fault Injection Techniques to the Experimental Assessment of the MARS Architecture

Johan Karlsson
Peter Folkesson
Chalmers University of Technology
Gothenburg, Sweden

Jean Arlat
Yves Crouzet
LAAS-CNRS
Toulouse, France

Günther Leber
Johannes Reisinger
Technical University of Vienna
Vienna, Austria

## Abstract

*This paper describes and compares three physical fault injection techniques—heavy-ion radiation, pin-level injection, and electromagnetic interference—and their use in the validation of MARS, a fault-tolerant distributed real-time system. The main features of the injection techniques are first summarized, and then the MARS system is described. The distributed testbed set-up and the common test scenario implemented to perform a coherent set of experiments by applying the three fault injection techniques are also described. The results are presented and discussed; special emphasis is put on the comparison of the specific impact of each technique.*

***Keywords:*** *Fault Tolerance, Coverage, Experimental Evaluation, Fault Injection, Physical Techniques*

## 1 Introduction

The dependability assessment of a fault-tolerant system is a complex task that requires the use of different levels of evaluation and related tools. Besides and in complement to other possible approaches such as proving or analytical modelling whose applicability and accuracy are significantly restricted in the case of complex fault-tolerant systems, *fault-injection* has been recognized to be particularly attractive and valuable. Indeed, by speeding up the occurrence of errors and failures, fault injection is in fact a method for *testing* the fault tolerance algorithms/ mechanisms with respect to their own specific inputs: *the faults*.

Fault injection can be applied either on a simulation model of the target fault-tolerant system (e.g., see [4, 7]) or on a hardware-and-software implementation (e.g., see [1, 26]).

Clearly simulation-based fault injection is desirable as it can provide early checks in the design process of fault tolerance algorithms/mechanisms. Nevertheless, it is worth noting that fault injection on a prototype featuring the actual interactions between the hardware and software dimensions of the fault tolerance algorithms/mechanisms supplies a more realistic and necessary complement to validate their implementation in a fault-tolerant system. Until recently, most studies related to the application of fault injection on a prototype of a fault-tolerant system relied on physical fault injection, i.e., the introduction of faults through the hardware layer of the target system [1, 6, 22]. A trend favouring the injection of errors through the software layer for simulating physical faults (i.e., software-implemented fault injection) has recently emerged (e.g., see [8, 23]). Although such an approach facilitates the application of fault injection, the correspondence between the types of errors that can be injected this way, and the actual faults is not yet confidently established. In spite of the difficulties in developing support environments and realizing experiments, physical fault injection enables real faults to be injected in a very close representation of the target system without any alteration to the software being executed.

Among the large number of experiments reported concerning physical fault injection, all used widely different techniques and/or were applied to distinct target systems. This significantly hampers the possibility to identify the difficulties/benefits associated to each fault injection technique and to analyse the results obtained.

This study relies on two major objectives. The first one is to get a better understanding of the impact and features of the three physical fault injection techniques that are considered and in which the sites have gained expertise in developing and applying dedicated experimental tools or in using standard support environments, respectively: heavy-ion radiation, pin-level injection and electromagnetic interferences (EMI). The distributed fault-tolerant system architecture MARS (Maintainable Real-Time System) developed by the Technical University of Vienna [10] is being used as the target system to carry out these experiments. Thus, the other driving objective is to evaluate the coverage of the built-in fault tolerance features of the MARS system. A distributed testbed architecture featuring five MARS nodes and a common test scenario have been

implemented at all three sites to perform a coherent set of experiments.

The remaining part of this paper is decomposed into six sections. Section 2 presents the main features of the fault injection techniques considered. The fault tolerance aspects of the MARS architecture and the structure of the MARS nodes are described in Section 3. Section 4 defines the approach considered for the experimental evaluation and the predicates characterizing the behaviour of the target system in the presence of injected faults. Section 5 depicts the common testbed set-up being used by all sites to carry out the experiments. The results are presented and discussed in Section 6. Finally, concluding remarks are provided in Section 7.

## 2 The Fault Injection Techniques

In this section, we briefly present the main features of the three fault injection techniques considered for the experimental assessment of the MARS system. Note that the pin-level and heavy-ion techniques have been largely reported in the literature, while the EMI technique has not previously been used for evaluation of error detection mechanisms.

### 2.1 Heavy-Ion Radiation

Heavy-ion radiation from a Californium-252 source can be used to inject single event upsets, i.e., bit flips at internal locations in integrated circuits. The heavy-ion method has been used to evaluate several hardware- and software-implemented error detection mechanisms for the MC6809E microprocessor [6, 15]. The irradiation of the target circuit must be performed in a vacuum as heavy-ions are attenuated by air molecules and other materials. Consequently, the packaging material that cover the target chip must also be removed. In these experiments, a miniature vacuum chamber containing the target circuit and the Cf-252 source was used. A comprehensive description of the heavy-ion fault injection technique and of the supporting tools is given in [9].

A major feature of the heavy-ion fault injection technique is that faults can be injected into VLSI circuits at locations which are impossible to reach by other techniques such as pin-level and software-implemented fault injection. The faults are also reasonably well spread within a circuit, as there are many sensitive memory elements in most VLSI circuits. Thereby, the injected faults generate a variety of error patterns which allows a thorough testing of fault handling mechanisms.

### 2.2 Pin-Level Injection

Pin-level fault injection, i.e., the injection of faults directly on the pins of the ICs of a prototype was until now the most widely applied physical fault injection technique. It has been used for (i) the evaluation of the coverage of specific mechanisms (in particular for error detection by means of signature analysis [22], and (ii) the validation of fault-tolerant distributed systems (e.g., [2, 26]). Flexible tools supporting some general features have been developed (e.g., the test facility used on the FTMP [13], MESSALINE at LAAS-CNRS [1] or RIFLE [14] at the University of Coimbra). The tool MESSALINE that will be used in these experiments is a flexible tool capable of adapting easily to various target systems and to different measures. It supports two implementations of pin-level fault injection:

- *forcing*, where the fault is directly applied by means of multi-pin probes on IC(s) pin(s) and associated equipotential line(s),
- *insertion*, where the IC(s) under test is(are) removed from the target system and inserted on a specific box where transistor switches ensure the proper isolation of the IC(s) under test from the system.

The fault models supported are stuck-at (0 or 1). Temporary faults can be injected on the pins of the ICs to simulate the consequences of such faults on the pins of the faulted IC(s).

### 2.3 EMI

An important class of computer failures are those caused by electro-magnetic interference (EMI). Such disturbances are common, for example, in motor cars, trains and industrial plants. Consequently, we decided to investigate the use of EMI for the evaluation of the MARS system. The fault injector used in the experiments generates bursts conforming to IEC 801-4 standard (CEI/IEC), i.e., the duration of the bursts is 15 ms, the period is 300 ms, the frequency is 1.25, 2.5, 5, or 10 kHz, and the voltage may be selected from 225 V to 4400 V (see Figure 1). These bursts are similar to those, which arise when switching inductive loads with relays or mechanical circuit-breakers.

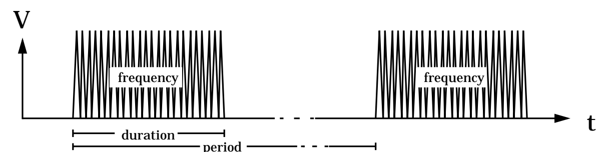


**Figure 1: Electro-magnetic bursts**

The faults were injected into the target system, which consisted of a single computer board, in two different ways (see Figure 2). In the first way, the computer board was placed between two conducting plates connected to the burst generator. The second way was to use a special probe that could expose a smaller part of the board to the

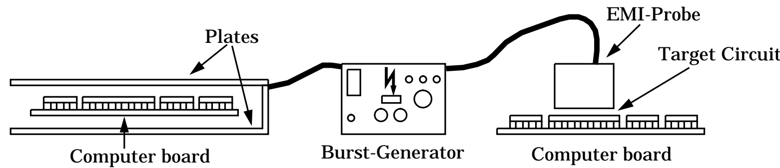


**Figure 2: Coupling of EMIs**

disturbances. In order to direct the faults to specific parts of the computer board, such a the CPU buses, small pieces of wire functioning as antennas were connected to the pins of specific ICs. The antennas were used with both the probe and the plates. In addition, experiments were also conducted using the probe without the antennas.

# 3 The MARS Architecture

This section summarizes the main fault tolerance features of the MARS architecture [10]. Fault tolerance issues at system-level are discussed first, then the structure of a special-purpose processing node designed to support these features in an optimal way are briefly described [21, 24]. Finally, special attention is paid to the identification and characterization of the error detection mechanisms.

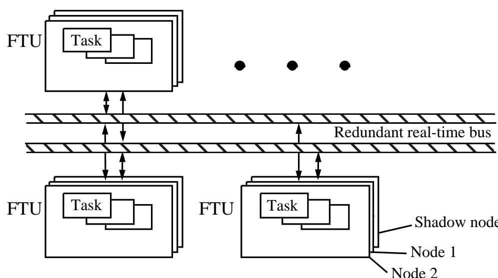


**Figure 3: System structure of MARS**

## 3.1 Fault Tolerance

A fault-tolerant distributed real-time system can be built up with a number of autonomous, fail-silent [17] processing nodes that are interconnected by a real-time network and which communicate by exchanging messages. The MARS system is the realization of such an approach. In MARS all active and passive components are replicated to prevent a single failure of such a component from causing a system failure. Up to three processing nodes execute identical software and form a Fault-Tolerant Unit (FTU). The FTUs communicate via the real-time network that is implemented as a replicated broadcast channel (see Figure 3).

MARS uses a two-layered mechanism to achieve fault-tolerance. The bottom layer (node layer) is responsible for error detection and error confinement (i.e., node shutdown on error). These functions are carried out by the processing nodes, which are therefore called “fail-silent”. The task of fail-silent nodes is to detect all internal errors and to prevent their propagation. Therefore, the top layer (system layer) needs not to care about erroneous data, it only has to provide enough redundancy to tolerate (silent) failures of parts of the system. The major functions of the top layer are handling of redundant data and reconfiguration of the system in case of a node failure [5].

To achieve a deterministic timing behaviour even in the presence of faults, the MARS system uses active redundancy for all processing and communication activities: each process is executed simultaneously at all nodes of an FTU and each message is transmitted quasi-simultaneously on each of the broadcast channels. Due to the fail-silence property, the results of all three nodes of an FTU are assumed to be correct and may be used interchangeably. Since we need only two nodes to tolerate a single failure of a fail-silent node (i.e., the loss of a message), the optional third node, the shadow node, does not transmit any message on the real-time network as long as both active nodes are operational. Only if an active node fails, the shadow node immediately starts to transmit its results, thus restoring the initial degree of redundancy.

## 3.2 Structure of the Processing Node

The single board implemented node consists of two independent processing units (see Figure 4), the application unit and the communication unit. Each unit is based on a 68070 CPU, a processor resembling the 68000 and including a memory management unit, a two-channel DMA controller, a UART (universal asynchronous receiver and transmitter) interface (RS232), an Inter-IC (I2C) bus, and an interrupt controller [16]. Further, each unit features a 16 bit parallel I/O-port and an EPROM. The application unit also contains a dynamic RAM, and two bidirectional FIFOs, one of which serves as an interface to external add-on hardware, the other one connects the application unit to the communication unit. Additional hardware for the communication unit comprises a Static Random Access Memory (SRAM), two Ethernet controllers, two Clock Synchronization Units (CSU) required for maintaining a global time base, and a watchdog timer.

## 3.3 The Error Detection Mechanisms

Three levels of error detection mechanisms (EDMs) are implemented in the MARS nodes: (i) the hardware EDMs, (ii) the system software EDMs implemented in the

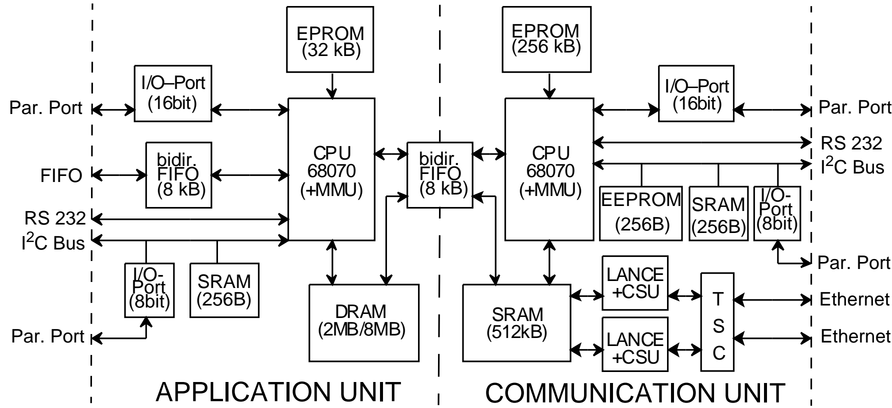


**Figure 4: Block diagram of the processor board**

operating system [11, 19, 20] and support software (i.e., the Modula/R compiler [25]), and (iii) the application level EDMs at the highest level. The error detection mechanisms provide the fail-silence property of the MARS nodes and are described in the following subsections.

**3.3.1 Hardware EDMs.** Whenever an error is detected by one of the hardware EDMs, in general, an exception is raised and the CPU will then wait for a reset issued by a watchdog timer. This watchdog timer is the only device, which may cause a reset of all devices including the CPU.

Two categories of hardware EDMs can be distinguished: the mechanisms provided by the CPU and those provided by special hardware on the processing board. In addition, faults can also trigger unexpected exceptions (i.e., neither the EDMs built into the CPU nor the mechanisms provided by special hardware are mapped to these exceptions).

The EDMs built into the CPU [16] are: bus error, address error, illegal op-code, privilege violation, division by zero, stack format error, uninitialized vector interrupt and spurious interrupt. These errors cause the processor to jump to the appropriate exception handling routines, which save the error state to a non volatile memory and then restart the node. Upon restart, a detailed error description is written to a serial port.

The following errors are detected by mechanisms implemented by special hardware on the node: silent shutdown of the CPU of the communication unit, power failure, parity error, FIFO over/underflow, access to physically non-existing memory, write access to the real-time network at an illegal point in time, error of an external device and error of the other unit. We call these “NMI mechanisms”, as they raise a Non-Maskable Interrupt when an error is detected.

An NMI leads to the same exception handling as the error detection mechanisms built into the CPU and can only be cleared by resetting the node, which is done by the watchdog timer.

**3.3.2 System Software EDMs.** The EDMs implemented by system software include mechanisms produced by the compiler (i.e., Compiler Generated Run-Time Assertions, CGRTA): value range overflow of a variable and loop iteration bound overflow.

The others are built into the operating system as assertions or as integrity checks on data: processing time overflow; various checks on data, done by the operating system; and various assertions coded into the operating system.

When an error is detected by any of these mechanisms, a trap instruction is executed, which leads to a node restart.

**3.3.3 Application Level EDMs.** The application level EDMs include end-to-end checksums for message data and double execution of tasks. The end-to-end checksums are used to detect mutilation of message data and is therefore used for implementing the extended fail-silence property of the nodes, i.e., the node is also considered to be fail-silent even when a corrupted message is sent, if the receiver detects the error and discards the message. Double execution of tasks in time redundancy can detect errors caused by transient faults that cause different output data of the two instances of the task. Combined with the concept of message checksums, task execution in time redundancy forms the highest level in the hierarchy of the error detection mechanisms. These mechanisms also trigger the execution of a trap instruction, which causes a reset of the node.

## 4 Measurements

The fail silence property of a MARS node when subjected to faults was assessed by means of fault injection campaigns using each of the techniques described in Section 2. In this section, we first provide an overview of the method supporting the experimental assessment, then we precisely define the predicates considered to perform the analyses. The common testbed set-up implemented for

carrying out the fault injection experiments with the three techniques is described in Section 5.

### 4.1 Experimental Assessment

Each campaign consists of several experiment runs. During each experiment a fault is injected into one node (node under test), another node (golden node) serves as a reference and a third node (comparator node) is used to compare the messages sent by the two previous nodes. Fault injection takes place until the node under test is declared to be failed by the comparator node. Then the node under test is shut down by the comparator node to clear all error conditions for the new experiment run. After some time, power is reinstalled and the node under test is reloaded for the next experiment run.

The assessment of the fail-silence property is obtained by monitoring the error detection information provided by EDMs of the node under test or by means of message checksum detections at the comparator node. Several combinations of enabled/disabled EDMs have been analysed, in order to study their impact on the fail silence property.

Although these measurements provide very valuable inputs for assessing the fail silence coverage of a MARS node, it is worth noting that estimating the 'real' coverage of the EDMs is a much more difficult task. The reason is that the real fault set usually is not known in detail, and even less is known about the probability of occurrence of the individual faults. In principle, an estimate of the 'real' coverage can be calculated as a weighted mean of the coverage factors obtained by different fault injection methods (e.g., see [18]). However, the lack of knowledge about the 'real' faults makes it very difficult—and in many practical cases impossible—to calculate the weight factors.

Each fault injection technique used here should therefore be considered strictly as a 'benchmark' method that can be used to evaluate the relative effectiveness of different EDMs. Combining several fault injection techniques improves the possibility to investigate coverage sensitivity with respect to changes in the error set.

### 4.2 Predicates

Four failure types can be distinguished for the node under test:

1) The node's EDMs detect an error and the node stops sending messages on the MARS bus; in this case the node stores the error condition into a non-volatile memory and resets itself by means of the watchdog timer.

2) The node fails to deliver the expected application message(s) for one or several application cycles, but no error is detected by the node's EDMs.

3) The node delivers a syntactically correct message with erroneous content. This is a fail-silence violation in the value domain, which is recognized as a mismatch between the messages sent by the node under test and the golden node.

4) The node sends a message at an illegal point in time, and thus disturbs the traffic on the MARS bus. This is a fail-silence violation in the time domain.

On every restart the node under test writes its previously saved error data, if available (i.e., if an error was detected by the node's EDMs), and data about its state to two serial ports, where it can be read and stored for further processing. From these data, the following predicates (events) can be derived:

Warmstart (*WS*): Warmstart (reset) of the node under test caused by the detection of (i) an error by the node's EDMs (*Internal WS*) or, (ii) an incoming or outgoing link failure by means of the top layer of the fault-tolerance mechanism, i.e., the membership protocol [10] (*External WS*).

Message loss (*ML*): One message (or more) from the node under test was lost (i.e., not received by the comparator node).

Message mismatch (*MM*): Reception by the comparator node of differing messages from golden node and tested node.

System Failure (*SF*): Failure of either the golden, data generation, or comparator nodes.

Coldstart (*CS*): Coldstart (power on) of the node under test is made after every experiment run, except when a system failure occurred.

The first four predicates roughly corresponds to the four failure types mentioned above. The *CS* predicate indicates the end of each data set. The assertion (occurrence) of the *WS* predicate in the data corresponds to the normal case when the node under test detects the error (failure type 1). The assertion of *ML* corresponds to a message loss failure (failure type 2); this behaviour is not a fail-silence violation, because no erroneous data is sent, but the error is not detected by the EDMs in the node under test. Irrespective of the other events, the assertion of *MM* (failure type 3) corresponds to a fail silence violation (in the data domain). There are two ways in which an *SF* failure may occur: (i) a fail silence violation in the time domain (failure type 4) affect the operation of the other nodes, or (ii) another node than the node under test experience a real hardware failure during the experiments. (Although, no *SF* failures were observed in the experiments, this failure type is described for the sake of completeness.)

Given the above failure types, the number of fail-silence violations can be counted as:

$$\#FS\ Viol.\ = \#Exp. \supseteq MM + \#Exp. \supseteq SF$$

where $\#\ Exp. \supseteq X$ counts the number of experiments where a $X$-type failure was diagnosed (i.e., predicate $X$ was asserted).

## 5 Common Experimental Set-up

The experimental set-up used by all sites consists of five MARS nodes and is similar to the one used in [3]. The workload is a realistic control application.

### 5.1 Test Application

As error detection coverage is highly dependent on the system activity, it is important to use a realistic workload in fault injection experiments. We selected a typical real-time application—a control problem—as workload in the MARS experiments. The control problem was taken from the rolling ball experiment [12] in which a ball is kept rolling along a circular path on a tiltable plane by controlling the two horizontal axes of the plane by servo motors and observing the position of the ball with a video camera. The tiltable plane and the camera are not present in our set-up; instead, the data from the camera is simulated by a data generation task. An additional task was provided, which compares the results of the two actively redundant computing nodes, both of which execute the control task.

The application executed during the fault injection experiments basically consists of three tasks (see also Figure 5):

1) The *data generation task* generates the input data for the control task. The input data include the nominal and actual values of the position, speed and acceleration of the ball.

2) The *control task*, which does not preserve any data or state information between its periodic executions, receives the emulated data from the data generation task and performs calculations on these data, i.e., calculates the desired acceleration for the ball.

3) The *comparator task* receives the results delivered by the two nodes that run the control-task in active redundancy, and compares them. This task also gives status information about the experiment, and assists in controlling the fault injection devices (i.e., it indicates when fault injection may take place) and the power supply of the node under test.

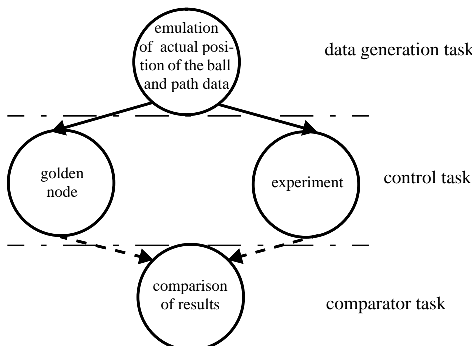


**Figure 5: Tasks and message flow**

The entire application has a period of 40 ms, i.e., all application tasks are started every 40 ms and hence produce a result in the same time interval. The application software is written in Modula/R [25] a Modula-2 like programming language with real time support for MARS.

### 5.2 Hardware Configuration

For the experiments five MARS nodes are needed (see Figure 6). One serves as a gateway between the department's local area network and the MARS-bus and is required for loading the entire application and for reloading the node under test. Another node executes the data generation task. The control task is performed on two actively working redundant nodes, one of which serves as a *golden node*, the other one is subjected to fault injection. The fifth node executes the comparator task. A UNIX workstation is used to control the experiments and to collect data for further analysis.

This experimental set-up is based on the assumption that the nodes are *replica determinate*, i.e., two replicated nodes produce always the same results if provided with the same input data. The MARS architecture supports this property.

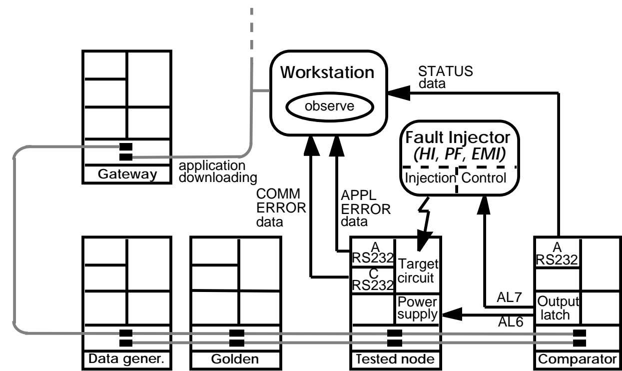


**Figure 6: Detailed set-up architecture**

### 5.3 Detailed Operation of the Experimental Set-up

Figure 6 describes the detailed set-up architecture and identifies the interactions with the fault injector devices. In the case of heavy-ion (HI), the target circuit is inserted in a miniature vacuum chamber containing a Cf-252 source; radiation can be controlled by an electrically manoeuvred shutter [9]. For pin-level injection, the pin-forcing (PF) technique is used; thus, the injection probe is directly connected to the pins of the target IC [1]. For EMI, both the technique using the two plates and the probe was used for the injections.

The experiments are controlled by the comparator MARS node and a UNIX workstation. The workstation is

also responsible for data collection. When the comparator node detects an error (error type ML or MM, see Section 4.2), it reports the error type to the workstation and turns off the power to the node under test with the signal AL 6. Signal AL 7 is used to discontinue fault injection (e.g., by closing the shutter mechanism of the vacuum chamber in the case of HI). Then the node under test is powered-up again and restarted. Upon restart, the application unit and the communication unit in the node under test send error data to the workstation via two serial lines. (If the error is not detected by the node under test itself, then the node has no error information available and sends only a status message). Once the node under test has been restarted, the workstation immediately starts to download the application. When the application has been restarted, the comparator node enables fault injection (signal AL 7) and a new experiment begins.

## 6 Results

The goal for the comparison of the fault injection techniques is to identify similarities and differences in the error sets generated by the three techniques. If the error sets are found to be disjoint, the fault injection techniques can be judged as fully complementary. In this case, applying all three techniques in the validation of a fault-tolerant system would improve the confidence of the validation results. In our case, the error sets were observed indirectly via the distribution of error detections among the various EDMs. To achieve as much similarity as possible among the error sets, faults were only injected inside, on the pins, or in the vicinity of either the application CPU or the communication CPU of the node under test.

Three different combinations of the application level EDMs have been evaluated for the three fault injection techniques considered. We use the following acronyms for these combinations: NOAM (no application level mechanisms, i.e., single execution and no checksums), SEMC (single execution, message checksums), DEMC (double execution, message checksums), see also Table 1. In addition a fourth combination, TEMC (triple execution, message checksums), was used in the heavy-ion experiments (see Section 6.1).

| Combination no. | Execution | Message Checksum | Acronym |
|---|---|---|---|
| 1 | Single | No | NOAM |
| 2 | Single | Yes | SEMC |
| 3 | Double | Yes | DEMC |
| 4 | Triple | Yes | TEMC |

**Table 1: Experimental combinations**

In the following paragraphs, we present in sequence the results obtained by the application of each technique. Then, these results are analysed and compared in a subsequent paragraph.

### 6.1 Heavy-Ion Radiation

Two circuits in the node under test were irradiated in separate experiments: the CPU of the application unit and the CPU of the communication unit. The irradiation was performed using a miniature vacuum chamber containing the irradiated circuit and a Cf-252 source (nominal activity 37 kBq); the distance between the source and the IC was approximately 35 mm. The IC's pin connections extended through the bottom plate of the miniature vacuum chamber, so that the chamber could be plugged directly into the socket of the irradiated IC in the MARS system.

Because the irradiated ICs were CMOS circuits, they had to be protected from heavy-ion induced latch-up. A latch-up is the triggering of a parasitic four layer switch (npnp or pnpn) acting as a silicon controlled rectifier (SCR), which may destroy the circuit due to excessive heat dissipation. The triggering of a latch-up is indicated by a drastic increase in the current drawn by the circuit. To prevent latch-ups from causing permanent damage to the ICs, a special device was used to turn off the power to the ICs when the current exceeded a threshold value.

Table 2 shows the distribution of error detections among the various EDMs for each of the irradiated CPUs, and the four combinations given in Table 1. The "Other" category in Table 2-a shows those errors for which no error information was given by the unit which contained the fault injected circuit. Error information was instead given by the other (fault free) unit of the tested node for some of these errors ("Other unit" category). This error information is detailed in Table 2-b. The "No error info." category gives the number of errors for which none of the two units in the MARS node produced error information.

The hardware EDMs, in particular the CPU mechanisms, detected most of the errors. This is not surprising since the faults were injected into the CPU. The proportion of errors detected by the hardware EDMs is larger for faults injected into the communication CPU than for faults injected into the application CPU. In particular, the coverage of the NMI EDMs is higher in the former case. Unexpected exceptions (UEE) occur with a frequency of about 15% in all combinations.

Errors detected by the OS EDMs dominate for the software EDMs, and for application level EDMs, the message checksum EDMs dominate.

The percentage of fail silence violations was between 2.4% and 0.5% for the NOAM, SEMC and DEMC combinations when faults were injected into the application CPU. As expected, the number of fail silence violations is lower for SEMC than for NOAM, and even lower for DEMC. When faults were injected into the communication CPU, only one fail silence violation was observed (for NOAM).

The observation of fail-silence violations for the

| Error Detection Mechanisms | | application unit CPU irradiated | | | | | | | | communication unit CPU irradiated | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | NOAM | | SEMC | | DEMC | | TEMC | | NOAM | | SEMC | | DEMC | |
| | | Errors | % | Errors | % | Errors | % | Errors | % | Errors | % | Errors | % | Errors | % |
| Level 1 Hardware | CPU | 3735 | 47.7% | 1410 | 49.0% | 4280 | 47.4% | 2573 | 51.3% | 1113 | 44.9% | 1270 | 43.2% | 1056 | 43.3% |
| | UEE | 1173 | 15.0% | 459 | 16.0% | 1373 | 15.2% | 736 | 14.7% | 361 | 14.6% | 416 | 14.1% | 326 | 13.4% |
| | NMI | 549 | 7.0% | 173 | 6.0% | 570 | 6.3% | 286 | 5.7% | 500 | 20.2% | 578 | 19.6% | 484 | 19.9% |
| | *Subtotal* | *5457* | *69.7%* | *2042* | *71.0%* | *6223* | *68.9%* | *3595* | *71.7%* | *1974* | *79.6%* | *2264* | *76.9%* | *1866* | *76.6%* |
| Level 2 Software | OS | 610 | 7.8% | 222 | 7.7% | 687 | 7.6% | 273 | 5.4% | 90 | 3.6% | 144 | 4.9% | 128 | 5.3% |
| | CGRTA | 75 | 1.0% | 3 | 0.1% | 30 | 0.3% | 37 | 0.7% | 10 | 0.4% | 7 | 0.2% | 13 | 0.5% |
| | *Subtotal* | *685* | *8.8%* | *225* | *7.8%* | *717* | *7.9%* | *310* | *6.2%* | *100* | *4.0%* | *151* | *5.1%* | *141* | *5.8%* |
| Level 3 Application level | Double exec. | — | — | — | — | 75 | 0.8% | 56 | 1.1% | — | — | — | — | 11 | 0.5% |
| | Checksum | — | — | 70 | 2.4% | 247 | 2.7% | 231 | 4.6% | — | — | 48 | 1.6% | 75 | 3.1% |
| | *Subtotal* | *—* | *—* | *70* | *2.4%* | *322* | *3.6%* | *287* | *5.7%* | *—* | *—* | *48* | *1.6%* | *86* | *3.5%* |
| Other | Other unit | 1095 | 14.0% | 381 | 13.2% | 1295 | 14.3% | 566 | 11.3% | 342 | 13.8% | 407 | 13.8% | 293 | 12.0% |
| | No error info. | 402 | 5.1% | 122 | 4.2% | 431 | 4.8% | 216 | 4.3% | 62 | 2.5% | 73 | 2.5% | 51 | 2.1% |
| | *Subtotal* | *1497* | *19.1%* | *503* | *17.5%* | *1726* | *19.1%* | *782* | *15.6%* | *404* | *16.3%* | *480* | *16.3%* | *344* | *14.1%* |
| Triple execution | | — | — | — | — | — | — | 42 | 0.8% | — | — | — | — | — | — |
| Fail silence violations | | 186 | 2.4% | 37 | 1.3% | 48 | 0.5% | 0 | 0% | 1 | <0.1% | 0 | 0% | 0 | 0% |
| *Total no. of errors* | | *7825* | *100%* | *2877* | *100%* | *9036* | *100%* | *5016* | *100%* | *2479* | *100%* | *2943* | *100%* | *2437* | *100%* |

**(a) Detection by the EDMs of the unit to which the faulted ICs belong**

| Error Detection Mechanisms | | application unit CPU irradiated | | | | | | | | communication unit CPU irradiated | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | NOAM | | SEMC | | DEMC | | TEMC | | NOAM | | SEMC | | DEMC | |
| | | Errors | % | Errors | % | Errors | % | Errors | % | Errors | % | Errors | % | Errors | % |
| Level 1 Hardware | CPU | 0 | 0% | 0 | 0% | 0 | 0% | 0 | 0% | 0 | 0% | 0 | 0% | 0 | 0% |
| | UEE | 3 | <0.1% | 2 | 0.1% | 0 | 0% | 3 | 0.1% | 0 | 0% | 0 | 0% | 1 | <0.1% |
| | NMI | 199 | 2.5% | 58 | 2.0% | 243 | 2.7% | 103 | 2.1% | 118 | 4.7% | 147 | 5.0% | 103 | 4.2% |
| | *Subtotal* | *202* | *2.6%* | *60* | *2.1%* | *243* | *2.7%* | *106* | *2.1%* | *118* | *4.7%* | *147* | *5.0%* | *104* | *4.3%* |
| Level 2 Software | OS | 893 | 11.4% | 321 | 11.2% | 1052 | 11.6% | 460 | 9.2% | 224 | 8.9% | 260 | 8.8% | 189 | 7.8% |
| | CGRTA | 0 | 0% | 0 | 0% | 0 | 0% | 0 | 0% | 0 | 0% | 0 | 0% | 0 | 0% |
| | *Subtotal* | *893* | *11.4%* | *321* | *11.2%* | *1052* | *11.6%* | *460* | *9.2%* | *224* | *8.9%* | *260* | *8.8%* | *189* | *7.8%* |
| Level 3 Application level | Double exec. | — | — | — | — | 0 | 0% | 0 | 0% | — | — | — | — | 0 | 0% |
| | Checksum | — | — | 0 | 0% | 0 | 0% | 0 | 0% | — | — | 0 | 0% | 0 | 0% |
| | *Subtotal* | *—* | *—* | *0* | *0%* | *0* | *0%* | *0* | *0%* | *—* | *—* | *0* | *0%* | *0* | *0%* |

**(b) Detection by the EDMs of the other unit (detail of "Other unit" entry in Table (a) above)**

**Table 2: Results for heavy-ion radiation**

DEMC combination was unexpected. In principle, all effects of transient faults should be masked by the double execution of tasks. One hypothesis for explaining these violations is that an undetected latch-up caused the same incorrect result to be produced by both executions of the control task.

To further investigate this hypothesis, experiments were carried out with the TEMC combination that used a third time redundant execution of the control task which was provided with fixed input data for which the results were known. This made it possible to detect errors by comparing the produced results with the correct results. This mechanism, which can be viewed as an on-line test program, would detect any semi-permanent fault such as the one suggested by the latch-up hypothesis.

The results show that no fail-silence violations occurred for the TEMC combination. As Table 2-a shows, 0.8% of the errors were detected by the third execution of the control task. This result supports the latch-up hypothesis. However, our experimental set-up does not provide sufficient observability to fully prove the latch-up hypothesis. In principle, the absence of fail-silence violations may merely be an effect of the change of the software configuration caused by the switch from DEMC to TEMC, and the errors detected by the third execution may have been caused by regular transients. Verification of the latch-up hypothesis, would require the use of a logic analyser so that the program flow and behaviour of the microprocessor could be studied in detail.

The OS and NMI EDMs dominate the detections made by the other unit of the tested node. The communications between the two units are carried out via two FIFO buffers, and nearly all of these detections are made by EDMs signalling empty FIFO. (An empty FIFO can be detected both by the operating system and the special NMI mechanism.)

## 6.2 Pin-Level Injection

The forcing technique was used for the fault injection experiments carried out on the MARS system. The main characteristics of the injected faults are listed hereafter:

- one single IC was fault injected at a time (the maximum number of pins faulted simultaneously — i.e., the multiplicity of the fault — being limited to $mx = 3$),
- uniform distribution over all combination of $mx$ pins was used to select the $mx$ faulted pins,
- stuck-at-0 and -1 fault models (all 0-1 combinations of $mx$ pins considered equally probable),
- to facilitate the comparison with the other techniques, both transient and intermittent (series of transients) faults were injected.

As the pin-forcing technique is being used, it can be

| Error Detection Mechanisms | | ICs belonging to the application unit | | | | | | ICs belonging to the communication unit | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | NOAM | | SEMC | | DEMC | | NOAM | | SEMC | | DEMC | |
| | | Errors | % | Errors | % | Errors | % | Errors | % | Errors | % | Errors | % |
| Level 1 Hardware | CPU | 71 | 11.2% | 53 | 9.0% | 38 | 7.0% | 37 | 6.9% | 37 | 8.2% | 20 | 3.9% |
| | UEE | 48 | 7.6% | 59 | 10.0% | 41 | 7.6% | 113 | 21.2% | 73 | 16.2% | 103 | 19.8% |
| | NMI | 474 | 75.0% | 430 | 73.0% | 423 | 78.2% | 265 | 49.7% | 260 | 57.5% | 263 | 50.7% |
| | *Subtotal* | *593* | *93.8%* | *542* | *92.0%* | *502* | *92.8%* | *415* | *77.9%* | *370* | *81.9%* | *386* | *74.4%* |
| Level 2 Software | OS | 6 | 0.9% | 6 | 1.0% | 7 | 1.3% | 35 | 6.6% | 21 | 4.6% | 30 | 5.8% |
| | CGRTA | 0 | 0% | 1 | 0.2% | 0 | 0% | 0 | 0% | 0 | 0% | 0 | 0% |
| | *Subtotal* | *6* | *0.9%* | *7* | *1.2%* | *7* | *1.3%* | *35* | *6.6%* | *21* | *4.6%* | *30* | *5.8%* |
| Level 3 Application level | Double exec. | — | — | — | — | 0 | 0% | — | — | — | — | 0 | 0% |
| | Checksum | — | — | 0 | 0% | 0 | 0% | — | — | 1 | 0.2% | 5 | 1.0% |
| | *Subtotal* | — | — | *0* | *0%* | *0* | *0%* | — | — | *1* | *0.2%* | *5* | *1.0%* |
| Other | Other unit | 1 | 0.2% | 8 | 1.4% | 2 | 0.4% | 23 | 4.3% | 17 | 3.8% | 26 | 5.0% |
| | No error info. | 32 | 5.1% | 30 | 5.1% | 30 | 5.5% | 59 | 11.1% | 43 | 9.5% | 72 | 13.9% |
| | *Subtotal* | *33* | *5.2%* | *38* | *6.5%* | *32* | *5.9%* | *82* | *15.4%* | *60* | *13.3%* | *98* | *18.9%* |
| Fail silence violations | | 0 | 0% | 2 | 0.3% | 0 | 0% | 1 | 0.2% | 0 | 0% | 0 | 0% |
| *Total no. of errors* | | *632* | *100%* | *589* | *100%* | *541* | *100%* | *533* | *100%* | *452* | *100%* | *519* | *100%* |

**(a) Detection by the EDMs of the unit to which the faulted ICs belong**

| Error Detection Mechanisms | | ICs belonging to the application unit | | | | | | ICs belonging to the communication unit | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | NOAM | | SEMC | | DEMC | | NOAM | | SEMC | | DEMC | |
| | | Errors | % | Errors | % | Errors | % | Errors | % | Errors | % | Errors | % |
| Level 1 Hardware | CPU | 0 | 0% | 0 | 0% | 0 | 0% | 0 | 0% | 0 | 0% | 0 | 0% |
| | UEE | 0 | 0% | 0 | 0% | 0 | 0% | 0 | 0% | 0 | 0% | 2 | 0.4% |
| | NMI | 1 | 0.2% | 7 | 1.2% | 2 | 0.4% | 23 | 4.3% | 17 | 3.8% | 24 | 4.6% |
| | *Subtotal* | *1* | *0.2%* | *7* | *1.2%* | *2* | *0.4%* | *23* | *4.3%* | *17* | *3.8%* | *26* | *5.0%* |
| Level 2 Software | OS | 0 | 0% | 1 | 0.2% | 0 | 0% | 0 | 0% | 0 | 0% | 0 | 0% |
| | CGRTA | 0 | 0% | 0 | 0% | 0 | 0% | 0 | 0% | 0 | 0% | 0 | 0% |
| | *Subtotal* | *0* | *0%* | *1* | *0.2%* | *0* | *0%* | *0* | *0%* | *0* | *0%* | *0* | *0%* |
| Level 3 Application level | Double exec. | — | — | — | — | 0 | 0% | — | — | — | — | 0 | 0% |
| | Checksum | — | — | 0 | 0% | 0 | 0% | — | — | 0 | 0% | 0 | 0% |
| | *Subtotal* | — | — | *0* | *0%* | *0* | *0%* | — | — | *0* | *0%* | *0* | *0%* |

**(b) Detection by the EDMs of the other unit (detail of “Other unit” entry in Table (a) above)**

**Table 3: Results for pin-level injection**

confidently considered that all pins of the ICs connected to an injected pin are tested as well. Accordingly, in the set of experiments conducted to date, to simplify the accessibility to the pins of the microprocessors of the application and communication units, the target ICs were mainly buffer ICs connected to them. Seven ICs (5 on the application unit and 2 on the communication unit) were tested. These tests resulted in a total of 3,266 error reports.

Table 3 shows the distribution of the errors detected by the various EDMs for the tested ICs of the tested node for three combinations of the application-level EDMs, together with their percentage of the total number of errors observed in each combination.

The results in Table 3-a indicate a dominant proportion of detections by the hardware EDMs (more than 90% on application unit side and 75% on communication unit side). NMI clearly dominates; however, in addition to CPU exceptions a significant number of UEEs were also triggered. The difference between UEE and NMI for the application and communication units can be explained by the fact that not all ICs tested on the application unit are directly connected to the processor. For software EDMs, detections by the OS significantly dominate. Concerning the application EDMs, the “Checksum” EDM mechanisms is dominating; no detections were triggered by the “Double execution” EDM when this option was enabled. Only a limited number of fail silence violations were observed: two occurrences for the SEMC combination when faults were injected on the application unit side.

Table 3-b shows that NMI error detection types are also dominating the supplementary detections observed on the other unit. Here also, a significant difference is observed between the results concerning the fault injections affecting the application unit side (from 0.2% to 1.2%) and the communication unit side (from 3.8% to 5.0%); this may indicate that a larger proportion of errors is propagated to the application unit when faults are injected in the communication unit rather than the converse.

## 6.3 EMI

Various fault-injection campaigns were carried out with a variety of voltage levels, with negative or positive polarity of the bursts, and with a burst-frequency of 2.5 kHz and 10 kHz. A total number of more than 17,000 errors were observed during all campaigns conducted with the first method, i.e., when the computer board of the node under test was mounted between two plates, and more than 30,000 errors were observed using the special probe (see Section 2.3). Most of the campaigns were conducted with all application level EDMs enabled.

In the first campaign shown in Table 4 (NOAM(1)) faults were injected into the communication unit using the two plates. Antenna wires were attached to the so-called

| Error Detection Mechanisms | | fault-injection with antennas | | | | | | fault-injection with probe only | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | NOAM(1) | | SEMC(2) | | DEMC(3) | | NOAM(4) | | SEMC(5) | | DEMC(6) | |
| | | Errors | % | Errors | % | Errors | % | Errors | % | Errors | % | Errors | % |
| Level 1 Hardware | CPU | 1195 | 72.0% | 193 | 76.6% | 137 | 2.2% | 4933 | 99.4% | 1692 | 98.1% | 1911 | 99.2% |
| | UEE | 11 | 0.7% | 8 | 3.2% | 9 | 0.2% | 31 | 0.6% | 17 | 1.0% | 15 | 0.8% |
| | NMI | 48 | 2.9% | 18 | 7.1% | 695 | 11.4% | 0 | 0% | 3 | 0.2% | 0 | 0% |
| | *Subtotal* | *1254* | *75.6%* | *219* | *86.9%* | *841* | *13.8%* | *4964* | *100%* | *1712* | *99.3%* | *1926* | *100%* |
| Level 2 Software | OS | 110 | 6.6% | 5 | 2.0% | 5215 | 85.6% | 0 | 0% | 3 | 0.2% | 0 | 0% |
| | CGRTA | 5 | 0.3% | 0 | 0% | 1 | <0.1% | 0 | 0% | 0 | 0% | 0 | 0% |
| | *Subtotal* | *115* | *6.9%* | *5* | *2.0%* | *5216* | *85.6%* | *0* | *0%* | *3* | *0.2%* | *0* | *0%* |
| Level 3 Application level | Double exec. | – | – | – | – | 9 | 0.2% | – | – | – | – | 0 | 0% |
| | Checksum | – | – | 1 | 0.4% | 8 | 0.1% | – | – | 1 | <0.1% | 0 | 0% |
| | *Subtotal* | *-* | *-* | *1* | *0.4%* | *17* | *0.3%* | *-* | *-* | *1* | *<0.1%* | *0* | *0%* |
| Other | Other unit | – | – | 24 | 9.5% | 6 | 0.1% | 0 | 0% | 6 | 0.3% | 0 | 0% |
| | No error info. | 271 | 16.3% | 0 | 0% | 13 | 0.2% | 0 | 0% | 2 | 0.1% | 0 | 0% |
| | *Subtotal* | *271* | *16.3%* | *24* | *9.5%* | *19* | *0.3%* | *0* | *0%* | *8* | *0.4%* | *0* | *0%* |
| Fail silence violations | | 20 | 1.2% | 3 | 1.2% | 0 | 0% | 0 | 0% | 0 | 0% | 0 | 0% |
| *Total number of errors* | | *1660* | *100%* | *252* | *100%* | *6093* | *100%* | *4964* | *100%* | *1724* | *100%* | *1926* | *100%* |

**(a) Detection by the EDMs of the unit to which the faulted ICs belong**

| Error Detection Mechanisms | | fault-injection with antennas | | | | | | fault-injection with probe only | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | NOAM(1) | | SEMC(2) | | DEMC(3) | | NOAM(4) | | SEMC(5) | | DEMC(6) | |
| | | Errors | % | Errors | % | Errors | % | Errors | % | Errors | % | Errors | % |
| Level 1 Hardware | CPU | – | – | 0 | 0% | 0 | 0% | 0 | 0% | 0 | 0% | 0 | 0% |
| | UEE | – | – | 0 | 0% | 0 | 0% | 0 | 0% | 0 | 0% | 0 | 0% |
| | NMI | – | – | 0 | 0% | 6 | 0.1% | 0 | 0% | 0 | 0% | 0 | 0% |
| | *Subtotal* | *–* | *–* | *0* | *0%* | *6* | *0.1%* | *0* | *0%* | *0* | *0%* | *0* | *0%* |
| Level 2 Software | OS | – | – | 24 | 9.5% | 0 | 0% | 0 | 0% | 6 | 0.3% | 0 | 0% |
| | CGRTA | – | – | 0 | 0% | 0 | 0% | 0 | 0% | 0 | 0% | 0 | 0% |
| | *Subtotal* | *–* | *–* | *24* | *9.5%* | *0* | *0%* | *0* | *0%* | *6* | *0.3%* | *0* | *0%* |
| Level 3 Application level | Double exec. | – | – | – | – | 0 | 0% | – | – | – | – | 0 | 0% |
| | Checksum | – | – | 0 | 0% | 0 | 0% | – | – | 0 | 0% | 0 | 0% |
| | *Subtotal* | *–* | *–* | *0* | *0%* | *0* | *0%* | *–* | *–* | *0* | *0%* | *0* | *0%* |

**(b) Detection by the EDMs of the other unit (detail of "Other unit" entry in Table (a) above)**

**Table 4: Results for EMI**

LO-EPROM, in order to disturb the address bus and the eight low order bits of the data bus. Bursts characterized by a frequency of 2.5 kHz, negative polarity, and a voltage of 230 V were injected. The second campaign (SEMC(2)) was conducted using the special probe with the wires connected to the corresponding LO-EPROM in the application unit. In this case the burst were characterized by a frequency of 10 kHz, negative polarity, and a voltage of 300 V. Campaign number three (DEMC(3)) used the two plates, the bursts had a frequency of 2.5 kHz, negative polarity, and voltage of 230 V. The wires were attached to the LO-EPROM of the application unit.

Campaigns 4 to 6 were only using the special probe for coupling faults into the CPU of the application unit, i.e., the probe was mounted on top of the CPU, and no wires were attached to any chip. The chosen frequency for the bursts was 10 kHz and negative polarity was used for all these experiments. We used a voltage of 290 V for campaign 4 and 6, while a slightly higher voltage, 300 V, was used for campaign 5.

Due to the large number of campaigns made, only selected campaigns are presented in Table 4, which shows the distribution of the errors detected by the various EDMs as total numbers and as percentage. Table 4-a shows the errors detected by the unit, where fault-injection was focused to; errors detected by the other unit of the node are detailed in Table 4-b.

Campaigns 1 and 2 show similar results, although focus of fault-injection was on different units of the processing node, the communication unit for campaign one and the application unit for campaign two. Most of the errors were detected by the hardware EDMs, where the CPU EDMs clearly dominate. For the software EDMs, which only detected a small fraction of the errors, the OS EDMs dominate. The relatively high amount of no error information for campaign one partly results from the fact that for this campaign no information about the errors detected by the application unit is available, because this is a result from early experiments, where only the outputs of the unit under test was recorded, and therefore all errors which were detected by the application unit are also counted as "No-error-info".

A different distribution of errors was observed for campaign three. There the software EDMs detected most of the errors, where error detection by OS EDMs dominates. Most of the errors detected by the OS EDMs were indicating that a message, that was required by the application, was lost. Note that campaign one and three both used the two plates, but the observed results are quite different. Campaigns 1 and 2 had different EMI conditions, but here the results are very similar. In general very different results were observed for similar conditions, e.g., slight changes in voltage levels. Thus, reproducibility appears to be problematic for EMI fault injection.

In campaigns 4 to 6 almost all of the errors were detected by the CPU EDMs. Only Campaign 5 shows a small amount of errors detected by other EDMs than hardware EDMs. When looking at the results of experiments 4 to 6 in more detail, which is not shown in Table 4 for brevity, we discovered that almost all of the detected errors were spurious interrupts detected by the processor. Spurious interrupts are interrupts signalled to the processor, but the processor cannot find the source of the interrupt, i.e., the device having raised the interrupt. This shows that the interrupt lines of a processor are highly sensitive to EMI.

Errors detected by the other unit were only detected by the NMI EDMs and by the OS EDMs in all campaigns.

### 6.4 Discussion

Almost all of the fault injection campaigns show that the hardware EDMs detect most of the errors. However, one campaign, EMI DEMC(3), shows results which are drastically different from the other campaigns. In order to simplify the discussion, we neglect the results from this campaign, when we compare the different results.

The main difference between the fault injection techniques, when looking at the hardware EDMs, is the number of errors detected by the CPU and the NMI mechanisms, respectively; the CPU EDMs dominate for heavy-ion radiation and EMI, while the NMI EDMs dominate for pin-forcing. A closer examination of the results showed that heavy-ion radiation exercised seven of the eight CPU EDMs, while EMI exercised five and pin-level exercised four of the CPU EDMs.

For EMI, when using the probe without antennas, the detection of spurious interrupts strongly dominated. Consequently, this method generates a very restricted error set, which clearly demonstrate that the method is not suitable for evaluation of error detection mechanisms. However, the variation in the error set was much larger when the antennas were used.

The proportion of unexpected exceptions is fairly large for pin-forcing and heavy-ion radiation, but quite low for the EMI technique.

Pin-forcing exercised 34 different combinations of NMI detections; the corresponding numbers for the heavy-ion and EMI techniques were 26 and 16, respectively. This indicates that pin-forcing may be more effective than the other techniques in exercising hardware EDMs located outside of the CPU chip.

One NMI mechanism of particular interest is the time-slice controller, which prevents access to the MARS bus at an illegal point in time. The results show that 5.0%, 11.6% and 1.9% of the errors were detected by the time-slice controller for heavy-ion, pin-forcing and EMI, respectively. Without this mechanism, the fail-silence property would have been violated in the time domain, which could lead to system failure (see Section 4.2). No fail-silence violations in the time domain were observed during the experiments.

The software EDMs detected the second largest amount of errors for all techniques. The unbalance observed in the case of heavy-ion radiation between the OS and CGRTA EDMs, is amplified when using pin-forcing and EMI: almost no detections by the CGRTAs were observed for the two latter techniques.

The application level EDMs detected the smallest amount of errors for all techniques, but when these were disabled, the fail-silence coverage was reduced (particularly for heavy-ion radiation) which shows the necessity of using these mechanisms as well.

The heavy-ion radiation stresses the system the most (i.e., the largest amount of fail-silence violations was observed for this technique). This technique also generates the largest error set, as indicated by the spread of the error detections among the EDMs. The spread of the detections is approximately the same for pin-forcing and EMI injections using antennas.

A limitation in this study is that the faults were injected only into two specific parts of the system—inside and around the two CPUs in the MARS node. The set-up also provided restricted observability as the effect of the errors could only be observed indirectly through the activations of the EDMs. Improved observability would allow more accurate comparisons, and can be achieved by incorporating a logic analyser in the experimental set-up, or combining physical experiments with simulation-based fault injections. Thus, more research is needed to further assess the relative merits and pitfalls of various fault injection techniques. Future studies should include other techniques such as software implemented fault injection and fault injection via boundary and internal scan chains.

## 7 Conclusion

This paper reported on a unique study devoted to the comparison of physical fault injection techniques. The paper described three techniques—heavy-ion radiation, pin-level fault injection, and EMI—and how they were used to validate the MARS system.

The results show fairly large differences in the distribution of the error detections among the various EDMs for the three fault injection techniques. This suggests that the techniques are rather complementary, i.e., they generate to a large extent different types of errors. The pin-forcing technique exercised the hardware EDMs located outside the CPU more effectively than the other techniques, while the heavy-ion and EMI techniques appear to be more suitable for exercising software and application level EDMs. Heavy-ion radiation showed the largest spread in the detections among the EDMs.

The validation of the MARS system showed that the

time-slice controller effectively prevents fail-silence violations in the time domain. Fail-silence violations in the value domain were observed for all three techniques when double time redundant execution of tasks was *not* used. In general, it was shown that the application level error detection mechanisms are necessary for improving fail-silence coverage.

## Acknowledgements

This work was partially supported by PDCS2 (Predictable Dependable Computing Systems 2), Basic Research Project No. 6362 of the ESPRIT programme. We thank Hermann Kopetz, the creator of the MARS system, for his support and many valuable suggestions. We are grateful to Lorenzo Strigini and Erland Jonsson for helpful comments on earlier versions of this paper.

## References

[1] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins and D. Powell, "Fault Injection for Dependability Validation — A Methodology and Some Applications", *IEEE Transactions on Software Engineering*, 16 (2), pp.166-82, February 1990.

[2] A. Damm, "The Effectiveness of Software Error-Detection Mechanisms in Real-Time Operating Systems", in *Proc. 16th Int. Symp. on Fault-Tolerant Computing (FTCS-16),* (Vienna, Austria), pp.171-6, IEEE Computer Society, 1986.

[3] A. Damm, *Experimental Evaluation of Error-detection and Self-Checking Coverage of Components of a Distributed Real-time System,* Doctoral Dissertation, Technical University, Vienna, Austria, October 1988.

[4] K. K. Goswami and R. K. Iyer, *DEPEND: A Simulation-Based Environment for System Level Dependability Analysis*, Univ. of Illinois at Urbana-Champaign, Tech. Report No. CHRC-92-11, June 1992.

[5] G. Grünsteidl and H. Kopetz, "A Reliable Multicast Protocol for Distributed Real-Time Systems", in *8th IEEE Workshop on Real-Time Operating Systems and Software,* (Atlanta, GA, USA), IEEE, 1991.

[6] U. Gunneflo, J. Karlsson and J. Torin, "Evaluation of Error Detection Schemes Using Fault Injection by Heavy-ion Radiation", in *Proc. 19th Int. Symp. Fault-Tolerant Computing (FTCS-19),* (Chicago, IL, USA), pp.340-7, IEEE Computer Society Press, 1989.

[7] E. Jenn, J. Arlat, M. Rimén, J. Ohlsson and J. Karlsson, "Fault Injection into VHDL Models: The MEFISTO Tool", in *Proc. 24th Int. Symp. on Fault-Tolerant Computing (FTCS-24),* (Austin, TX, USA), pp.66-75, IEEE Computer Society Press, 1994.

[8] G. A. Kanawati, N. A. Kanawati and J. A. Abraham, "FERRARI: A Tool for the Validation of System Dependability Properties", in *Proc. 22nd. Int. Symp. on Fault-Tolerant Computing (FTCS-22),* (Boston, MA, USA), pp.336-44, IEEE Computer Society Press, 1992.

[9] J. Karlsson, P. Lidén, P. Dahlgren and R. Johansson, "Using Heavy-Ion Radiation to Validate Fault-Handling Mechanisms", *IEEE Micro*, 14 (1), pp.8-23, February 1994.

[10] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft and R. Zainlinger, "Distributed Fault-Tolerant Real-Time Systems: The MARS Approach", *IEEE Micro*, 9 (1), pp.25-40, February 1989.

[11] H. Kopetz, G. Fohler, G. Grünsteidl, H. Kantz, G. Pospischil, P. Puschner, J. Reisinger, R. Schlatterbeck, W. Schütz, A. Vrchoticky and R. Zainlinger, *"The Distributed Fault-Tolerant Real-Time Operating System MARS",* IEEE Operating Systems Newsletter, 6 (1), 1992.

[12] H. Kopetz, P. Holzer, G. Leber and M. Schindler, *The Rolling Ball on MARS*, Institut für Technische Informatik, Technical University, Vienna, Tech. Report No. 13/91, 1991.

[13] J. H. Lala, "Fault Detection, Isolation, and Reconfiguration in FTMP: Methods and Experimental Results", in *Fifth AIAA/IEEE Digital Avionics Sys. Conf.,* pp.21.3.1-.3.9, 1983.

[14] H. Madeira, M. Rela, F. Moreira and J. G. Silva, "A General Purpose Pin-level Fault Injector", in *Proc. 1st European Dependable Computing Conf. (EDCC-1),* (Berlin, Germany), pp.199-216, Springer-Verlag, 1994.

[15] G. Miremadi, J. Karlsson, U. Gunneflo and J. Torin, "Two Software Techniques for On-line Error Detection", in *Proc. 22nd Int. Symp. on Fault-Tolerant Computing (FTCS-22),* (Boston, MA, USA), pp.328-44, IEEE Computer Society Press, 1992.

[16] Philips Semiconductors, *SCC68070 User Manual 1991, Part 1 - Hardware,* 1992.

[17] D. Powell, G. Bonn, D. Seaton, P. Veríssimo and F. Waeselynck, "The Delta-4 Approach to Dependability in Open Distributed Computing Systems", in *Proc. 18th Int. Symp. on Fault-Tolerant Computing Systems (FTCS-18),* (Tokyo, Japan), pp.246-51, IEEE Computer Society Press, 1988.

[18] D. Powell, E. Martins, J. Arlat and Y. Crouzet, "Estimators for Fault Tolerance Coverage Evaluation", *IEEE Transactions on Computers*, 44 (2), pp.261-74, February 1995.

[19] J. Reisinger, "Time Driven Operating Systems - A Case Study on the MARS Kernel", in *5th ACM SIGOPS European Workshop,* (Le Mont Saint Michel, France), 1992.

[20] J. Reisinger, *Conception and Analysis of a Time Triggered Operating System for Real-Time Applications,* Doctoral Dissertation, Technical University, Vienna, July, 1993, in German.

[21] J. Reisinger and A. Steininger, "The Design of a Fail-Silent Processing Node for MARS", *Distributed Systems Engineering Journal*, 1 (1), pp.104-11, 1993.

[22] M. A. Schuette, J. P. Shen, D. P. Siewiorek and Y. X. Zhu, "Experimental Evaluation of Two Concurrent Error Detection Schemes", in *Proc. 16th Int. Symp. Fault-Tolerant Computing (FTCS-16),* (Vienna, Austria), pp.138-43, IEEE Computer Society Press, 1986.

[23] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, D. Rancey, A. Robinson and T. Lin, "FIAT — Fault Injection-based Automated Testing Environment", in *Proc. 18th Int. Symp. on Fault-Tolerant Computing (FTCS-18),* (Tokyo, Japan), pp.102-7, IEEE Computer Society Press, 1988.

[24] A. Steininger and J. Reisinger, "Integral Design of Hardware and Operating System for a DCCS", in *10th IFAC Workshop on Distributed Computer Control Systems,* (Semmering, Austria), Pergamon Press, 1991.

[25] A. Vrchoticky, *Modula/R Language Definition*, Institut für Technische Informatik, Technical University, Vienna, Tech. Report No. 2/92, 1992.

[26] C. J. Walter, "Evaluation and Design of an Ultra-Reliable Distributed Architecture for Fault Tolerance", *IEEE Transactions on Reliability*, 39 (4), pp.492-9, October 1990.